Programming Language—Common Lisp

9. Conditions

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

9.1 Condition System Concepts

Common Lisp constructs are described not only in terms of their behavior in situations during which they are intended to be used (see the "Description" part of each *operator* specification), but in all other situations (see the "Exceptional Situations" part of each *operator* specification).

A situation is the evaluation of an expression in a specific context. A *condition* is an *object* that represents a specific situation that has been detected. *Conditions* are *generalized instances* of the *class* **condition**. A hierarchy of *condition* classes is defined in Common Lisp. A *condition* has *slots* that contain data relevant to the situation that the *condition* represents.

An error is a situation in which normal program execution cannot continue correctly without some form of intervention (either interactively by the user or under program control). Not all errors are detected. When an error goes undetected, the effects can be *implementation-dependent*, *implementation-defined*, unspecified, or undefined. See Section 1.4 (Definitions). All detected errors can be represented by *conditions*, but not all *conditions* represent errors.

Signaling is the process by which a *condition* can alter the flow of control in a program by raising the *condition* which can then be *handled*. The functions **error**, **cerror**, **signal**, and **warn** are used to signal *conditions*.

The process of signaling involves the selection and invocation of a *handler* from a set of *active handlers*. A *handler* is a *function* of one argument (the *condition*) that is invoked to handle a *condition*. Each *handler* is associated with a *condition type*, and a *handler* will be invoked only on a *condition* of the *handler*'s associated *type*.

Active handlers are *established* dynamically (see **handler-bind** or **handler-case**). *Handlers* are invoked in a *dynamic environment* equivalent to that of the signaler, except that the set of *active handlers* is bound in such a way as to include only those that were *active* at the time the *handler* being invoked was *established*. Signaling a *condition* has no side-effect on the *condition*, and there is no dynamic state contained in a *condition*.

If a *handler* is invoked, it can address the *situation* in one of three ways:

Decline

It can decline to *handle* the *condition*. It does this by simply returning rather than transferring control. When this happens, any values returned by the handler are ignored and the next most recently established handler is invoked. If there is no such handler and the signaling function is **error** or **cerror**, the debugger is entered in the *dynamic environment* of the signaler. If there is no such handler and the signaling function is either **signal** or **warn**, the signaling function simply returns **nil**.

Handle

It can *handle* the *condition* by performing a non-local transfer of control. This can be done either primitively by using **go**, **return**, **throw** or more abstractly by using a function such as **abort** or **invoke-restart**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

Defer

It can put off a decision about whether to *handle* or *decline*, by any of a number of actions, but most commonly by signaling another condition, resignaling the same condition, or forcing entry into the debugger.

9.1.1 Condition Types

Figure 9–1 lists the *standardized condition types*. Additional *condition types* can be defined by using **define-condition**.

arithmetic-error	**floating-point-overflow**	**simple-type-error**
cell-error	**floating-point-underflow**	**simple-warning**
condition	**package-error**	**storage-condition**
control-error	**parse-error**	**stream-error**
division-by-zero	**print-not-readable**	**style-warning**
end-of-file	**program-error**	**type-error**
error	**reader-error**	**unbound-slot**
file-error	**serious-condition**	**unbound-variable**
floating-point-inexact	**simple-condition**	**undefined-function**
floating-point-invalid-operation	**simple-error**	**warning**

Figure 9–1. Standardized Condition Types

All *condition* types are *subtypes* of *type* **condition**. That is,

```
(typep c 'condition) → true
```

if and only if `c` is a *condition*.

Implementations must define all specified *subtype* relationships. Except where noted, all *subtype* relationships indicated in this document are not mutually exclusive. A *condition* inherits the structure of its *supertypes*.

The metaclass of the *class* **condition** is not specified. *Names* of *condition types* may be used to specify *supertype* relationships in **define-condition**, but the consequences are not specified if an attempt is made to use a *condition type* as a *superclass* in a **defclass** *form*.

Figure 9–2 shows *operators* that define *condition types* and creating *conditions*.

define-condition	**make-condition**

Figure 9–2. Operators that define and create conditions.

Figure 9–3 shows *operators* that *read* the *value* of *condition slots*.

arithmetic-error-operands	**simple-condition-format-arguments**
arithmetic-error-operation	**simple-condition-format-control**
cell-error-name	**stream-error-stream**
file-error-pathname	**type-error-datum**
package-error-package	**type-error-expected-type**
print-not-readable-object	**unbound-slot-instance**

Figure 9–3. Operators that read condition slots.

9.1.1.1 Serious Conditions

A *serious condition* is a *condition* serious enough to require interactive intervention if not handled. *Serious conditions* are typically signaled with **error** or **cerror**; non-serious *conditions* are typically signaled with **signal** or **warn**.

9.1.2 Creating Conditions

The function **make-condition** can be used to construct a *condition object* explicitly. Functions such as **error**, **cerror**, **signal**, and **warn** operate on *conditions* and might create *condition objects* implicitly. Macros such as **ccase**, **ctypecase**, **ecase**, **etypecase**, **check-type**, and **assert** might also implicitly create (and *signal*) *conditions*.

9.1.2.1 Condition Designators

A number of the functions in the condition system take arguments which are identified as ***condition designators***. By convention, those arguments are notated as

datum `&rest` *arguments*

Taken together, the *datum* and the *arguments* are "*designators* for a *condition* of default type *default-type*." How the denoted *condition* is computed depends on the type of the *datum*:

- If the *datum* is a *symbol* naming a *condition type* ...

  The denoted *condition* is the result of

  ```
  (apply #'make-condition datum arguments)
  ```


Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

- If the *datum* is a *format control* ...

  The denoted *condition* is the result of

  ```
  (make-condition defaulted-type
                  :format-control datum
                  :format-arguments arguments)
  ```

  where the *defaulted-type* is a *subtype* of *default-type*.

- If the *datum* is a *condition* ...

  The denoted *condition* is the *datum* itself. In this case, unless otherwise specified by the description of the *operator* in question, the *arguments* must be *null*; that is, the consequences are undefined if any *arguments* were supplied.

Note that the *default-type* gets used only in the case where the *datum string* is supplied. In the other situations, the resulting condition is not necessarily of *type* *default-type*.

Here are some illustrations of how different *condition designators* can denote equivalent *condition objects*:

```
(let ((c (make-condition 'arithmetic-error :operator '/ :operands '(7 0))))
  (error c))
≡ (error 'arithmetic-error :operator '/ :operands '(7 0))

(error "Bad luck.")
≡ (error 'simple-error :format-control "Bad luck." :format-arguments '())
```

9.1.3 Printing Conditions

If the `:report` argument to **define-condition** is used, a print function is defined that is called whenever the defined *condition* is printed while the *value* of ***print-escape*** is *false*. This function is called the ***condition reporter***; the text which it outputs is called a ***report message***.

When a *condition* is printed and ***print-escape*** is *false*, the *condition reporter* for the *condition* is invoked. *Conditions* are printed automatically by functions such as **invoke-debugger**, **break**, and **warn**.

When ***print-escape*** is *true*, the *object* should print in an abbreviated fashion according to the style of the implementation (*e.g.*, by **print-unreadable-object**). It is not required that a *condition* can be recreated by reading its printed representation.

No *function* is provided for directly *accessing* or invoking *condition reporters*.

9.1.3.1 Recommended Style in Condition Reporting

In order to ensure a properly aesthetic result when presenting *report messages* to the user, certain stylistic conventions are recommended.

There are stylistic recommendations for the content of the messages output by *condition reporters*, but there are no formal requirements on those *programs*. If a *program* violates the recommendations for some message, the display of that message might be less aesthetic than if the guideline had been observed, but the *program* is still considered a *conforming program*.

The requirements on a *program* or *implementation* which invokes a *condition reporter* are somewhat stronger. A *conforming program* must be permitted to assume that if these style guidelines are followed, proper aesthetics will be maintained. Where appropriate, any specific requirements on such routines are explicitly mentioned below.

9.1.3.1.1 Capitalization and Punctuation in Condition Reports

It is recommended that a *report message* be a complete sentences, in the proper case and correctly punctuated. In English, for example, this means the first letter should be uppercase, and there should be a trailing period.

```
(error "This is a message")  ; Not recommended
(error "this is a message.") ; Not recommended

(error "This is a message.") ; Recommended instead
```

9.1.3.1.2 Leading and Trailing Newlines in Condition Reports

It is recommended that a *report message* not begin with any introductory text, such as "`Error: `" or "`Warning: `" or even just *freshline* or *newline*. Such text is added, if appropriate to the context, by the routine invoking the *condition reporter*.

It is recommended that a *report message* not be followed by a trailing *freshline* or *newline*. Such text is added, if appropriate to the context, by the routine invoking the *condition reporter*.

```
(error "This is a message.~%")   ; Not recommended
(error "~&This is a message.")   ; Not recommended
(error "~&This is a message.~%") ; Not recommended

(error "This is a message.")     ; Recommended instead
```

9.1.3.1.3 Embedded Newlines in Condition Reports

Especially if it is long, it is permissible and appropriate for a *report message* to contain one or more embedded *newlines*.

If the calling routine conventionally inserts some additional prefix (such as "`Error: `" or "`;; Error: `") on the first line of the message, it must also assure that an appropriate prefix

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

will be added to each subsequent line of the output, so that the left edge of the message output by the *condition reporter* will still be properly aligned.

```
(defun test ()
  (error "This is an error message.~%It has two lines."))

;; Implementation A
(test)
This is an error message.
It has two lines.

;; Implementation B
(test)
;; Error: This is an error message.
;;        It has two lines.

;; Implementation C
(test)
>> Error: This is an error message.
          It has two lines.
```

9.1.3.1.4 Note about Tabs in Condition Reports

Because the indentation of a *report message* might be shifted to the right or left by an arbitrary amount, special care should be taken with the semi-standard *character* ⟨*Tab*⟩ (in those *implementations* that support such a *character*). Unless the *implementation* specifically defines its behavior in this context, its use should be avoided.

9.1.3.1.5 Mentioning Containing Function in Condition Reports

The name of the containing function should generally not be mentioned in *report messages*. It is assumed that the *debugger* will make this information accessible in situations where it is necessary and appropriate.

9.1.4 Signaling and Handling Conditions

The operation of the condition system depends on the ordering of active *applicable handlers* from most recent to least recent.

Each *handler* is associated with a *type specifier* that must designate a *subtype* of *type* **condition**. A *handler* is said to be *applicable* to a *condition* if that *condition* is of the *type* designated by the associated *type specifier*.

Active handlers are *established* by using **handler-bind** (or an abstraction based on **handler-bind**, such as **handler-case** or **ignore-errors**).

Active handlers can be *established* within the dynamic scope of other *active handlers*. At any point during program execution, there is a set of *active handlers*. When a *condition* is signaled, the *most recent* active *applicable handler* for that *condition* is selected from this set. Given a *condition*, the order of recentness of active *applicable handlers* is defined by the following two rules:

1. Each handler in a set of active handlers H_1 is more recent than every handler in a set H_2 if the handlers in H_2 were active when the handlers in H_1 were established.

2. Let h_1 and h_2 be two applicable active handlers established by the same *form*. Then h_1 is more recent than h_2 if h_1 was defined to the left of h_2 in the *form* that established them.

Once a handler in a handler binding *form* (such as **handler-bind** or **handler-case**) has been selected, all handlers in that *form* become inactive for the remainder of the signaling process. While the selected *handler* runs, no other *handler* established by that *form* is active. That is, if the *handler* declines, no other handler established by that *form* will be considered for possible invocation.

Figure 9–4 shows *operators* relating to the *handling* of *conditions*.

handler-bind	**handler-case**	**ignore-errors**

Figure 9–4. Operators relating to handling conditions.

9.1.4.1 Signaling

When a *condition* is signaled, the most recent applicable *active handler* is invoked. Sometimes a handler will decline by simply returning without a transfer of control. In such cases, the next most recent applicable active handler is invoked.

If there are no applicable handlers for a *condition* that has been signaled, or if all applicable handlers decline, the *condition* is unhandled.

The functions **cerror** and **error** invoke the interactive *condition* handler (the debugger) rather than return if the *condition* being signaled, regardless of its *type*, is unhandled. In contrast, **signal** returns **nil** if the *condition* being signaled, regardless of its *type*, is unhandled.

The *variable* ***break-on-signals*** can be used to cause the debugger to be entered before the signaling process begins.

Figure 9–5 shows *defined names* relating to the *signaling* of *conditions*.

break-on-signals	**error**	**warn**
cerror	**signal**	

Figure 9–5. Defined names relating to signaling conditions.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

9.1.4.1.1 Resignaling a Condition

During the *dynamic extent* of the *signaling* process for a particular *condition object*, **signaling** the same *condition object* again is permitted if and only if the *situation* represented in both cases are the same.

For example, a *handler* might legitimately *signal* the *condition object* that is its *argument* in order to allow outer *handlers* first opportunity to *handle* the condition. (Such a *handlers* is sometimes called a "default handler.") This action is permitted because the *situation* which the second *signaling* process is addressing is really the same *situation*.

On the other hand, in an *implementation* that implemented asynchronous keyboard events by interrupting the user process with a call to **signal**, it would not be permissible for two distinct asynchronous keyboard events to *signal identical condition objects* at the same time for different situations.

9.1.4.2 Restarts

The interactive condition handler returns only through non-local transfer of control to specially defined *restarts* that can be set up either by the system or by user code. Transferring control to a restart is called "invoking" the restart. Like handlers, active *restarts* are *established* dynamically, and only active *restarts* can be invoked. An active *restart* can be invoked by the user from the debugger or by a program by using **invoke-restart**.

A *restart* contains a *function* to be *called* when the *restart* is invoked, an optional name that can be used to find or invoke the *restart*, and an optional set of interaction information for the debugger to use to enable the user to manually invoke a *restart*.

The name of a *restart* is used by **invoke-restart**. *Restarts* that can be invoked only within the debugger do not need names.

Restarts can be established by using **restart-bind**, **restart-case**, and **with-simple-restart**. A *restart* function can itself invoke any other *restart* that was active at the time of establishment of the *restart* of which the *function* is part.

The *restarts established* by a **restart-bind** *form*, a **restart-case** *form*, or a **with-simple-restart** *form* have *dynamic extent* which extends for the duration of that *form*'s execution.

Restarts of the same name can be ordered from least recent to most recent according to the following two rules:

1. Each *restart* in a set of active restarts R_1 is more recent than every *restart* in a set R_2 if the *restarts* in R_2 were active when the *restarts* in R_1 were established.

2. Let r_1 and r_2 be two active *restarts* with the same name established by the same *form*. Then r_1 is more recent than r_2 if r_1 was defined to the left of r_2 in the *form* that established them.

If a *restart* is invoked but does not transfer control, the values resulting from the *restart* function are returned by the function that invoked the restart, either **invoke-restart** or **invoke-restart-interactively**.

9.1.4.2.1 Interactive Use of Restarts

For interactive handling, two pieces of information are needed from a *restart*: a report function and an interactive function.

The report function is used by a program such as the debugger to present a description of the action the *restart* will take. The report function is specified and established by the `:report-function` keyword to **restart-bind** or the `:report` keyword to **restart-case**.

The interactive function, which can be specified using the `:interactive-function` keyword to **restart-bind** or `:interactive` keyword to **restart-case**, is used when the *restart* is invoked interactively, such as from the debugger, to produce a suitable list of arguments.

invoke-restart invokes the most recently *established restart* whose name is the same as the first argument to **invoke-restart**. If a *restart* is invoked interactively by the debugger and does not transfer control but rather returns values, the precise action of the debugger on those values is *implementation-defined*.

9.1.4.2.2 Interfaces to Restarts

Some *restarts* have functional interfaces, such as **abort**, **continue**, **muffle-warning**, **store-value**, and **use-value**. They are ordinary functions that use **find-restart** and **invoke-restart** internally, that have the same name as the *restarts* they manipulate, and that are provided simply for notational convenience.

Figure 9–6 shows *defined names* relating to *restarts*.

abort	**invoke-restart-interactively**	**store-value**
compute-restarts	**muffle-warning**	**use-value**
continue	**restart-bind**	**with-simple-restart**
find-restart	**restart-case**	
invoke-restart	**restart-name**	

Figure 9–6. Defined names relating to restarts.

9.1.4.2.3 Restart Tests

Each *restart* has an associated test, which is a function of one argument (a *condition* or **nil**) which returns *true* if the *restart* should be visible in the current *situation*. This test is created by the `:test-function` option to **restart-bind** or the `:test` option to **restart-case**.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

9.1.4.2.4 Associating a Restart with a Condition

A *restart* can be "associated with" a *condition* explicitly by **with-condition-restarts**, or implicitly by **restart-case**. Such an assocation has *dynamic extent*.

A single *restart* may be associated with several *conditions* at the same time. A single *condition* may have several associated *restarts* at the same time.

Active restarts associated with a particular *condition* can be detected by *calling* a *function* such as **find-restart**, supplying that *condition* as the *condition argument*. Active restarts can also be detected without regard to any associated *condition* by calling such a function without a *condition argument*, or by supplying a value of **nil** for such an *argument*.

9.1.5 Assertions

Conditional signaling of *conditions* based on such things as key match, form evaluation, and *type* are handled by assertion *operators*. Figure 9–7 shows *operators* relating to assertions.

assert	**check-type**	**ecase**
ccase	**ctypecase**	**etypecase**

Figure 9–7. Operators relating to assertions.

9.1.6 Notes about the Condition System's Background

For a background reference to the abstract concepts detailed in this section, see *Exceptional Situations in Lisp*. The details of that paper are not binding on this document, but may be helpful in establishing a conceptual basis for understanding this material.

condition *Condition Type*

Class Precedence List:

condition, **t**

Description:

All types of *conditions*, whether error or non-error, must inherit from this *type*.

No additional *subtype* relationships among the specified *subtypes* of *type* **condition** are allowed, except when explicitly mentioned in the text; however implementations are permitted to introduce additional *types* and one of these *types* can be a *subtype* of any number of the *subtypes* of *type* **condition**.

Whether a user-defined *condition type* has *slots* that are accessible by *with-slots* is *implementation-dependent*. Furthermore, even in an *implementation* in which user-defined *condition types* would have *slots*, it is *implementation-dependent* whether any *condition types* defined in this document have such *slots* or, if they do, what their *names* might be; only the reader functions documented by this specification may be relied upon by portable code.

Conforming code must observe the following restrictions related to *conditions*:

- **define-condition**, not **defclass**, must be used to define new *condition types*.
- **make-condition**, not **make-instance**, must be used to create *condition objects* explicitly.
- The `:report` option of **define-condition**, not **defmethod** for **print-object**, must be used to define a condition reporter.
- **slot-value**, **slot-boundp**, **slot-makunbound**, and **with-slots** must not be used on *condition objects*. Instead, the appropriate accessor functions (defined by **define-condition**) should be used.

warning *Condition Type*

Class Precedence List:

warning, **condition**, **t**

Description:

The *type* **warning** consists of all types of warnings.

See Also:

style-warning

style-warning *Condition Type*

Class Precedence List:

style-warning, **warning**, **condition**, **t**

Description:

The *type* **style-warning** includes those *conditions* that represent *situations* involving *code* that is *conforming code* but that is nevertheless considered to be faulty or substandard.

See Also:

muffle-warning

Notes:

An *implementation* might signal such a *condition* if it encounters *code* that uses deprecated features or that appears unaesthetic or inefficient.

An 'unused variable' warning must be of *type* **style-warning**.

In general, the question of whether *code* is faulty or substandard is a subjective decision to be made by the facility processing that *code*. The intent is that whenever such a facility wishes to complain about *code* on such subjective grounds, it should use this *condition type* so that any clients who wish to redirect or muffle superfluous warnings can do so without risking that they will be redirecting or muffling other, more serious warnings.

serious-condition *Condition Type*

Class Precedence List:

serious-condition, **condition**, **t**

Description:

All *conditions* serious enough to require interactive intervention if not handled should inherit from the *type* **serious-condition**. This condition type is provided primarily so that it may be included as a *superclass* of other *condition types*; it is not intended to be signaled directly.

Notes:

Signaling a *serious condition* does not itself force entry into the debugger. However, except in the unusual situation where the programmer can assure that no harm will come from failing to *handle* a *serious condition*, such a *condition* is usually signaled with **error** rather than **signal** in order to assure that the program does not continue without *handling* the *condition*. (And conversely, it is conventional to use **signal** rather than **error** to signal conditions which are not *serious conditions*, since normally the failure to handle a non-serious condition is not reason enough for the debugger to be entered.)

error *Condition Type*

Class Precedence List:

error, **serious-condition**, **condition**, **t**

Description:

The *type* **error** consists of all *conditions* that represent *errors*.

cell-error *Condition Type*

Class Precedence List:

cell-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **cell-error** consists of error conditions that occur during a location *access*. The name of the offending cell is initialized by the `:name` initialization argument to **make-condition**, and is *accessed* by the *function* **cell-error-name**.

See Also:

cell-error-name

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

cell-error-name *Function*

Syntax:

cell-error-name *condition* → *name*

Arguments and Values:

condition—a *condition* of *type* **cell-error**.

name—an *object*.

Description:

Returns the *name* of the offending cell involved in the *situation* represented by *condition*.

The nature of the result depends on the specific *type* of *condition*. For example, if the *condition* is of *type* **unbound-variable**, the result is the *name* of the *unbound variable* which was being *accessed*, if the *condition* is of *type* **undefined-function**, this is the *name* of the *undefined function* which was being *accessed*, and if the *condition* is of *type* **unbound-slot**, this is the *name* of the *slot* which was being *accessed*.

See Also:

cell-error, **unbound-slot**, **unbound-variable**, **undefined-function**, Section 9.1 (Condition System Concepts)

parse-error *Condition Type*

Class Precedence List:

parse-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **parse-error** consists of error conditions that are related to parsing.

See Also:

parse-namestring, **reader-error**

storage-condition *Condition Type*

Class Precedence List:

storage-condition, **serious-condition**, **condition**, **t**

Description:

The *type* **storage-condition** consists of serious conditions that relate to problems with memory management that are potentially due to *implementation-dependent* limits rather than semantic errors in *conforming programs*, and that typically warrant entry to the debugger if not handled. Depending on the details of the *implementation*, these might include such problems as stack overflow, memory region overflow, and storage exhausted.

Notes:

While some Common Lisp operations might signal *storage-condition* because they are defined to create *objects*, it is unspecified whether operations that are not defined to create *objects* create them anyway and so might also signal **storage-condition**. Likewise, the evaluator itself might create *objects* and so might signal **storage-condition**. (The natural assumption might be that such *object* creation is naturally inefficient, but even that is *implementation-dependent*.) In general, the entire question of how storage allocation is done is *implementation-dependent*, and so any operation might signal **storage-condition** at any time. Because such a *condition* is indicative of a limitation of the *implementation* or of the *image* rather than an error in a *program*, *objects* of *type* **storage-condition** are not of *type* **error**.

assert *Macro*

Syntax:

assert *test-form* [({*place*}*) [*datum-form* {*argument-form*}*]]
→ **nil**

Arguments and Values:

test-form—a *form*; always evaluated.

place—a *place*; evaluated if an error is signaled.

datum-form—a *form* that evaluates to a *datum*. Evaluated each time an error is to be signaled, or not at all if no error is to be signaled.

argument-form—a *form* that evaluates to an *argument*. Evaluated each time an error is to be signaled, or not at all if no error is to be signaled.

datum, *arguments*—*designators* for a *condition* of default type **error**. (These *designators* are the result of evaluating *datum-form* and each of the *argument-forms*.)

assert

Description:

assert assures that *test-form* evaluates to *true*. If *test-form* evaluates to *false*, **assert** signals a *correctable error* (denoted by *datum* and *arguments*). Continuing from this error using the **continue** *restart* makes it possible for the user to alter the values of the *places* before **assert** evaluates *test-form* again. If the value of *test-form* is *non-nil*, **assert** returns **nil**.

The *places* are *generalized references* to data upon which *test-form* depends, whose values can be changed by the user in attempting to correct the error. *Subforms* of each *place* are only evaluated if an error is signaled, and might be re-evaluated if the error is re-signaled (after continuing without actually fixing the problem). The order of evaluation of the *places* is not specified; see Section 5.1.1.1 (Evaluation of Subforms to Places). If a *place form* is supplied that produces more values than there are store variables, the extra values are ignored. If the supplied *form* produces fewer values than there are store variables, the missing values are set to **nil**.

Examples:

```
 (setq x (make-array '(3 5) :initial-element 3))
→ #2A((3 3 3 3 3) (3 3 3 3 3) (3 3 3 3 3))
 (setq y (make-array '(3 5) :initial-element 7))
→ #2A((7 7 7 7 7) (7 7 7 7 7) (7 7 7 7 7))
 (defun matrix-multiply (a b)
   (let ((*print-array* nil))
     (assert (and (= (array-rank a) (array-rank b) 2)
                  (= (array-dimension a 1) (array-dimension b 0)))
             (a b)
             "Cannot multiply ~S by ~S." a b)
            (really-matrix-multiply a b))) → MATRIX-MULTIPLY
 (matrix-multiply x y)
▷ Correctable error in MATRIX-MULTIPLY:
▷ Cannot multiply #<ARRAY ...> by #<ARRAY ...>.
▷ Restart options:
▷  1: You will be prompted for one or more new values.
▷  2: Top level.
▷ Debug> :continue 1
▷ Value for A: x
▷ Value for B: (make-array '(5 3) :initial-element 6)
→ #2A((54 54 54 54 54)
      (54 54 54 54 54)
      (54 54 54 54 54)
      (54 54 54 54 54)
      (54 54 54 54 54))

 (defun double-safely (x) (assert (numberp x) (x)) (+ x x))
 (double-safely 4)
→ 8
```

```
 (double-safely t)
▷ Correctable error in DOUBLE-SAFELY: The value of (NUMBERP X) must be non-NIL.
▷ Restart options:
▷  1: You will be prompted for one or more new values.
▷  2: Top level.
▷ Debug> :continue 1
▷ Value for X: 7
→ 14
```

Affected By:

break-on-signals

The set of active *condition handlers*.

See Also:

check-type, **error**, Section 5.1 (Generalized Reference)

Notes:

The debugger need not include the *test-form* in the error message, and the *places* should not be included in the message, but they should be made available for the user's perusal. If the user gives the "continue" command, the values of any of the references can be altered. The details of this depend on the implementation's style of user interface.

error *Function*

Syntax:

error *datum* &rest *arguments* → |

Arguments and Values:

datum, *arguments*—*designators* for a *condition* of default type **simple-error**.

Description:

error effectively invokes **signal** on the denoted *condition*.

If the *condition* is not handled, (invoke-debugger *condition*) is done. As a consequence of calling **invoke-debugger**, **error** cannot directly return; the only exit from **error** can come by non-local transfer of control in a handler or by use of an interactive debugging command.

Examples:

```
 (defun factorial (x)
   (cond ((or (not (typep x 'integer)) (minusp x))
          (error "~S is not a valid argument to FACTORIAL." x))
         ((zerop x) 1)
```



```
         (t (* x (factorial (- x 1))))))
→ FACTORIAL
(factorial 20)
→ 2432902008176640000
(factorial -1)
▷ Error: -1 is not a valid argument to FACTORIAL.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Return to Lisp Toplevel.
▷ Debug>

 (setq a 'fred)
→ FRED
 (if (numberp a) (1+ a) (error "~S is not a number." A))
▷ Error: FRED is not a number.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Return to Lisp Toplevel.
▷ Debug> :Continue 1
▷ Return to Lisp Toplevel.

 (define-condition not-a-number (error)
                   ((argument :reader not-a-number-argument :initarg :argument))
   (:report (lambda (condition stream)
              (format stream "~S is not a number."
                      (not-a-number-argument condition)))))
→ NOT-A-NUMBER

 (if (numberp a) (1+ a) (error 'not-a-number :argument a))
▷ Error: FRED is not a number.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Return to Lisp Toplevel.
▷ Debug> :Continue 1
▷ Return to Lisp Toplevel.
```

Side Effects:

Handlers for the specified condition, if any, are invoked and might have side effects. Program execution might stop, and the debugger might be entered.

Affected By:

Existing handler bindings.

break-on-signals

Signals an error of *type* **type-error** if *datum* and *arguments* are not *designators* for a *condition*.

See Also:

cerror, **signal**, **format**, **ignore-errors**, ***break-on-signals***, **handler-bind**, Section 9.1 (Condition

System Concepts)

Notes:

Some implementations may provide debugger commands for interactively returning from individual stack frames. However, it should be possible for the programmer to feel confident about writing code like:

```
(defun wargames:no-win-scenario ()
  (if (error "pushing the button would be stupid."))
  (push-the-button))
```

In this scenario, there should be no chance that **error** will return and the button will get pushed.

While the meaning of this program is clear and it might be proven 'safe' by a formal theorem prover, such a proof is no guarantee that the program is safe to execute. Compilers have been known to have bugs, computers to have signal glitches, and human beings to manually intervene in ways that are not always possible to predict. Those kinds of errors, while beyond the scope of the condition system to formally model, are not beyond the scope of things that should seriously be considered when writing code that could have the kinds of sweeping effects hinted at by this example.

cerror *Function*

Syntax:

cerror *continue-format-control datum* &rest *arguments* → **nil**

Arguments and Values:

Continue-format-control—a *format control*.

datum, *arguments*—*designators* for a *condition* of default type **simple-error**.

Description:

cerror effectively invokes **error** on the *condition* named by *datum*. As with any function that implicitly calls **error**, if the *condition* is not handled, (invoke-debugger *condition*) is executed. While signaling is going on, and while in the debugger if it is reached, it is possible to continue code execution (*i.e.*, to return from **cerror**) using the **continue** *restart*.

If *datum* is a *condition*, *arguments* can be supplied, but are used only in conjunction with the *continue-format-control*.

Examples:

```
(defun real-sqrt (n)
  (when (minusp n)
    (setq n (- n))
```



```
    (cerror "Return sqrt(~D) instead." "Tried to take sqrt(-~D)." n))
  (sqrt n))

(real-sqrt 4)
→ 2.0

(real-sqrt -9)
▷ Correctable error in REAL-SQRT: Tried to take sqrt(-9).
▷ Restart options:
▷  1: Return sqrt(9) instead.
▷  2: Top level.
▷ Debug> :continue 1
→ 3.0

(define-condition not-a-number (error)
  ((argument :reader not-a-number-argument :initarg :argument))
  (:report (lambda (condition stream)
             (format stream "~S is not a number."
                     (not-a-number-argument condition)))))

(defun assure-number (n)
  (loop (when (numberp n) (return n))
        (cerror "Enter a number."
                'not-a-number :argument n)
        (format t "~&Type a number: ")
        (setq n (read))
        (fresh-line)))

(assure-number 'a)
▷ Correctable error in ASSURE-NUMBER: A is not a number.
▷ Restart options:
▷  1: Enter a number.
▷  2: Top level.
▷ Debug> :continue 1
▷ Type a number: 1/2
→ 1/2

(defun assure-large-number (n)
  (loop (when (and (numberp n) (> n 73)) (return n))
        (cerror "Enter a number~:[~; a bit larger than ~D~]."
                "~*~A is not a large number."
                (numberp n) n)
        (format t "~&Type a large number: ")
        (setq n (read))
        (fresh-line)))
```

```
 (assure-large-number 10000)
→ 10000

 (assure-large-number 'a)
▷ Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
▷ Restart options:
▷  1: Enter a number.
▷  2: Top level.
▷ Debug> :continue 1
▷ Type a large number: 88
→ 88

 (assure-large-number 37)
▷ Correctable error in ASSURE-LARGE-NUMBER: 37 is not a large number.
▷ Restart options:
▷  1: Enter a number a bit larger than 37.
▷  2: Top level.
▷ Debug> :continue 1
▷ Type a large number: 259
→ 259

 (define-condition not-a-large-number (error)
   ((argument :reader not-a-large-number-argument :initarg :argument))
   (:report (lambda (condition stream)
              (format stream "~S is not a large number."
                      (not-a-large-number-argument condition)))))

 (defun assure-large-number (n)
   (loop (when (and (numberp n) (> n 73)) (return n))
         (cerror "Enter a number~3*~:[~; a bit larger than ~*~D~]."
                 'not-a-large-number
                 :argument n
                 :ignore (numberp n)
                 :ignore n
                 :allow-other-keys t)
         (format t "~&Type a large number: ")
         (setq n (read))
         (fresh-line)))


 (assure-large-number 'a)
▷ Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
▷ Restart options:
▷  1: Enter a number.
▷  2: Top level.
▷ Debug> :continue 1
▷ Type a large number: 88
→ 88

 (assure-large-number 37)
▷ Correctable error in ASSURE-LARGE-NUMBER: A is not a large number.
▷ Restart options:
▷  1: Enter a number a bit larger than 37.
▷  2: Top level.
▷ Debug> :continue 1
▷ Type a large number: 259
→ 259
```


Affected By:

break-on-signals.

Existing handler bindings.

See Also:

error, **format**, **handler-bind**, ***break-on-signals***, **simple-type-error**

Notes:

If *datum* is a *condition type* rather than a *string*, the **format** directive `~*` may be especially useful in the *continue-format-control* in order to ignore the *keywords* in the *initialization argument list*. For example:

```
(cerror "enter a new value to replace ~*~s"
        'not-a-number
        :argument a)
```

check-type *Macro*

Syntax:

check-type *place typespec* [*string*] → **nil**

Arguments and Values:

place—a *place*.

typespec—a *type specifier*.

string—a *string*; evaluated.

Description:

check-type signals a *correctable error* of *type* **type-error** if the contents of *place* are not of the type *typespec*.

check-type can return only if the **store-value** *restart* is invoked, either explicitly from a handler or implicitly as one of the options offered by the debugger. If the **store-value** *restart* is invoked, **check-type** stores the new value that is the argument to the *restart* invocation (or that is prompted for interactively by the debugger) in *place* and starts over, checking the type of the new value and signaling another error if it is still not of the desired *type*.

The first time *place* is *evaluated*, it is *evaluated* by normal evaluation rules. It is later *evaluated* as a *place* if the type check fails and the **store-value** *restart* is used; see Section 5.1.1.1 (Evaluation of Subforms to Places).

string should be an English description of the type, starting with an indefinite article ("a" or "an"). If *string* is not supplied, it is computed automatically from *typespec*. The automatically generated message mentions *place*, its contents, and the desired type. An implementation may choose to generate a somewhat differently worded error message if it recognizes that *place* is of a particular form, such as one of the arguments to the function that called **check-type**. *string* is allowed because some applications of **check-type** may require a more specific description of what is wanted than can be generated automatically from *typespec*.

Examples:

```
 (setq aardvarks '(sam harry fred))
→ (SAM HARRY FRED)
 (check-type aardvarks (array * (3)))
▷ Error: The value of AARDVARKS, (SAM HARRY FRED),
▷        is not a 3-long array.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use instead.
▷  2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Use Value: #(SAM FRED HARRY)
→ NIL
 aardvarks
→ #<ARRAY-T-3 13571>
 (map 'list #'identity aardvarks)
→ (SAM FRED HARRY)
 (setq aardvark-count 'foo)
→ FOO
 (check-type aardvark-count (integer 0 *) "A positive integer")
▷ Error: The value of AARDVARK-COUNT, FOO, is not a positive integer.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use instead.
▷  2: Top level.
```



```
▷ Debug> :CONTINUE 2

 (defmacro define-adder (name amount)
   (check-type name (and symbol (not null)) "a name for an adder function")
   (check-type amount integer)
   `(defun ,name (x) (+ x ,amount)))

 (macroexpand '(define-adder add3 3))
→ (defun add3 (x) (+ x 3))

 (macroexpand '(define-adder 7 7))
▷ Error: The value of NAME, 7, is not a name for an adder function.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use instead.
▷  2: Top level.
▷ Debug> :Continue 1
▷ Specify a value to use instead.
▷ Type a form to be evaluated and used instead: 'ADD7
→ (defun add7 (x) (+ x 7))

 (macroexpand '(define-adder add5 something))
▷ Error: The value of AMOUNT, SOMETHING, is not an integer.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use instead.
▷  2: Top level.
▷ Debug> :Continue 1
▷ Type a form to be evaluated and used instead: 5
→ (defun add5 (x) (+ x 5))
```

Control is transferred to a handler.

Side Effects:

The debugger might be entered.

Affected By:

break-on-signals

The implementation.

See Also:

Section 9.1 (Condition System Concepts)

Notes:

```
 (check-type place typespec)
```

```
≡ (assert (typep place 'typespec) (place)
          'type-error :datum place :expected-type 'typespec)
```

simple-error *Condition Type*

Class Precedence List:

simple-error, **simple-condition**, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **simple-error** consists of *conditions* that are signaled by **error** or **cerror** when a *format control* is supplied as the function's first argument.

invalid-method-error *Function*

Syntax:

invalid-method-error *method format-control* &rest *args* → *implementation-dependent*

Arguments and Values:

method—a *method*.

format-control—a *format control*.

args—*format arguments* for the *format-control*.

Description:

The *function* **invalid-method-error** is used to signal an error of *type* **error** when there is an applicable *method* whose *qualifiers* are not valid for the method combination type. The error message is constructed by using the *format-control* suitable for **format** and any *args* to it. Because an implementation may need to add additional contextual information to the error message, **invalid-method-error** should be called only within the dynamic extent of a method combination function.

The *function* **invalid-method-error** is called automatically when a *method* fails to satisfy every *qualifier* pattern and predicate in a **define-method-combination** *form*. A method combination function that imposes additional restrictions should call **invalid-method-error** explicitly if it encounters a *method* it cannot accept.

Whether **invalid-method-error** returns to its caller or exits via **throw** is *implementation-dependent*.

Side Effects:

The debugger might be entered.

Affected By:

break-on-signals

See Also:

define-method-combination

method-combination-error *Function*

Syntax:

method-combination-error *format-control* &rest *args* → *implementation-dependent*

Arguments and Values:

format-control—a *format control*.

args—*format arguments* for *format-control*.

Description:

The *function* **method-combination-error** is used to signal an error in method combination.

The error message is constructed by using a *format-control* suitable for **format** and any *args* to it. Because an implementation may need to add additional contextual information to the error message, **method-combination-error** should be called only within the dynamic extent of a method combination function.

Whether **method-combination-error** returns to its caller or exits via **throw** is *implementation-dependent*.

Side Effects:

The debugger might be entered.

Affected By:

break-on-signals

See Also:

define-method-combination

signal *Function*

Syntax:

signal *datum* &rest *arguments* → **nil**

Arguments and Values:

datum, *arguments*—*designators* for a *condition* of default type **simple-condition**.

Description:

Signals the *condition* denoted by the given *datum* and *arguments*. If the *condition* is not handled, **signal** returns **nil**.

Examples:

```
 (defun handle-division-conditions (condition)
   (format t "Considering condition for division condition handling~%")
   (when (and (typep condition 'arithmetic-error)
              (eq '/ (arithmetic-error-operation condition)))
     (invoke-debugger condition)))
HANDLE-DIVISION-CONDITIONS
 (defun handle-other-arithmetic-errors (condition)
   (format t "Considering condition for arithmetic condition handling~%")
   (when (typep condition 'arithmetic-error)
     (abort)))
HANDLE-OTHER-ARITHMETIC-ERRORS
 (define-condition a-condition-with-no-handler (condition) ())
A-CONDITION-WITH-NO-HANDLER
 (signal 'a-condition-with-no-handler)
NIL
 (handler-bind ((condition #'handle-division-conditions)
                  (condition #'handle-other-arithmetic-errors))
   (signal 'a-condition-with-no-handler))
Considering condition for division condition handling
Considering condition for arithmetic condition handling
NIL
 (handler-bind ((arithmetic-error #'handle-division-conditions)
                  (arithmetic-error #'handle-other-arithmetic-errors))
   (signal 'arithmetic-error :operation '* :operands '(1.2 b)))
Considering condition for division condition handling
Considering condition for arithmetic condition handling
Back to Lisp Toplevel
```

Side Effects:

The debugger might be entered due to ***break-on-signals***.

Handlers for the condition being signaled might transfer control.

Affected By:

Existing handler bindings.

break-on-signals

See Also:

break-on-signals, **error**, **simple-condition**, Section 9.1.4 (Signaling and Handling Conditions)

Notes:

If (typep *datum* *break-on-signals*) *yields true*, the debugger is entered prior to beginning the signaling process. The **continue** *restart* can be used to continue with the signaling process. This is also true for all other *functions* and *macros* that should, might, or must *signal conditions*.

simple-condition *Condition Type*

Class Precedence List:

simple-condition, **condition**, **t**

Description:

The *type* **simple-condition** represents *conditions* that are signaled by **signal** whenever a *format-control* is supplied as the function's first argument. The *format control* and *format arguments* are initialized with the initialization arguments named :format-control and :format-arguments to **make-condition**, and are *accessed* by the *functions* **simple-condition-format-control** and **simple-condition-format-arguments**. If format arguments are not supplied to **make-condition**, **nil** is used as a default.

See Also:

simple-condition-format-control, **simple-condition-format-arguments**

simple-condition-format-control, simple-condition-format-arguments *Function*

Syntax:

simple-condition-format-control *condition* → *format-control*

simple-condition-format-arguments *condition* → *format-arguments*

Arguments and Values:

condition—a *condition* of *type* **simple-condition**.

format-control—a *format control*.

format-arguments—a *list*.

Description:

simple-condition-format-control returns the *format control* needed to process the *condition*'s *format arguments*.

simple-condition-format-arguments returns a *list* of *format arguments* needed to process the *condition*'s *format control*.

Examples:

```
 (setq foo (make-condition 'simple-condition
                          :format-control "Hi ~S"
                          :format-arguments '(ho)))
→ #<SIMPLE-CONDITION 26223553>
 (apply #'format nil (simple-condition-format-control foo)
                     (simple-condition-format-arguments foo))
→ "Hi HO"
```

See Also:

simple-condition, Section 9.1 (Condition System Concepts)

warn *Function*

Syntax:

warn *datum* &rest *arguments* → **nil**

Arguments and Values:

datum, *arguments*—*designators* for a *condition* of default type **simple-warning**.

Description:

Signals a *condition* of *type* **warning**. If the *condition* is not *handled*, reports the *condition* to *error output*.

The precise mechanism for warning is as follows:

The warning condition is signaled

While the **warning** *condition* is being signaled, the **muffle-warning** *restart* is established for use by a *handler*. If invoked, this *restart* bypasses further action by **warn**, which in turn causes **warn** to immediately return **nil**.

If no handler for the warning condition is found

If no handlers for the warning condition are found, or if all such handlers decline, then the *condition* is reported to *error output* by **warn** in an *implementation-dependent* format.

nil is returned

The value returned by **warn** if it returns is **nil**.

Examples:

```
  (defun foo (x)
    (let ((result (* x 2)))
      (if (not (typep result 'fixnum))
          (warn "You're using very big numbers."))
      result))
→ FOO

  (foo 3)
→ 6

  (foo most-positive-fixnum)
▷ Warning: You're using very big numbers.
→ 4294967294

  (setq *break-on-signals* t)
→ T

  (foo most-positive-fixnum)
▷ Break: Caveat emptor.
▷ To continue, type :CONTINUE followed by an option number.
▷  1: Return from Break.
▷  2: Abort to Lisp Toplevel.
▷ Debug> :continue 1
```

```
▷ Warning: You're using very big numbers.
→ 4294967294
```

Side Effects:

A warning is issued. The debugger might be entered.

Affected By:

Existing handler bindings.

break-on-signals, ***error-output***.

Exceptional Situations:

If *datum* is a *condition* and if the *condition* is not of *type* **warning**, or *arguments* is *non-nil*, an error of *type* **type-error** is signaled.

If *datum* is a condition type, the result of `(apply #'make-condition datum arguments)` must be of *type* **warning** or an error of *type* **type-error** is signaled.

See Also:

break-on-signals, **muffle-warning**, **signal**

simple-warning *Condition Type*

Class Precedence List:

simple-warning, **simple-condition**, **warning**, **condition**, **t**

Description:

The *type* **simple-warning** represents *conditions* that are signaled by **warn** whenever a *format control* is supplied as the function's first argument.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

invoke-debugger *Function*

Syntax:

invoke-debugger *condition* →|

Arguments and Values:

condition—a *condition object*.

Description:

invoke-debugger attempts to enter the debugger with *condition*.

If ***debugger-hook*** is not **nil**, it should be a *function* (or the name of a *function*) to be called prior to entry to the standard debugger. The *function* is called with ***debugger-hook*** bound to **nil**, and the *function* must accept two arguments: the *condition* and the *value* of ***debugger-hook*** prior to binding it to **nil**. If the *function* returns normally, the standard debugger is entered.

The standard debugger never directly returns. Return can occur only by a non-local transfer of control, such as the use of a restart function.

Examples:

```
 (ignore-errors ;Normally, this would suppress debugger entry
   (handler-bind ((error #'invoke-debugger)) ;But this forces debugger entry
     (error "Foo.")))
Debug: Foo.
To continue, type :CONTINUE followed by an option number:
 1: Return to Lisp Toplevel.
Debug>
```

Side Effects:

debugger-hook is bound to **nil**, program execution is discontinued, and the debugger is entered.

Affected By:

debug-io and ***debugger-hook***.

See Also:

error, **break**

break *Function*

Syntax:

break &optional *format-control* &rest *format-arguments* → **nil**

Arguments and Values:

format-control—a *format control*. The default is *implementation-dependent*.

format-arguments—*format arguments* for the *format-control*.

Description:

break *formats* *format-control* and *format-arguments* and then goes directly into the debugger without allowing any possibility of interception by programmed error-handling facilities.

If the **continue** *restart* is used while in the debugger, **break** immediately returns **nil** without taking any unusual recovery action.

break binds ***debugger-hook*** to **nil** before attempting to enter the debugger.

Examples:

```
 (break "You got here with arguments: ~:S." '(FOO 37 A))
▷ BREAK: You got here with these arguments: FOO, 37, A.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Return from BREAK.
▷  2: Top level.
▷ Debug> :CONTINUE 1
▷ Return from BREAK.
→ NIL
```

Side Effects:

The debugger is entered.

Affected By:

debug-io.

See Also:

error, **invoke-debugger**.

Notes:

break is used as a way of inserting temporary debugging "breakpoints" in a program, not as a way of signaling errors. For this reason, **break** does not take the *continue-format-control argument* that **cerror** takes. This and the lack of any possibility of interception by *condition handling* are the only program-visible differences between **break** and **cerror**.

The user interface aspects of **break** and **cerror** are permitted to vary more widely, in order to accomodate the interface needs of the *implementation*. For example, it is permissible for a *Lisp read-eval-print loop* to be entered by **break** rather than the conventional debugger.

break could be defined by:

```
 (defun break (&optional (format-control "Break") &rest format-arguments)
   (with-simple-restart (continue "Return from BREAK.")
     (let ((*debugger-hook* nil))
       (invoke-debugger
           (make-condition 'simple-condition
                           :format-control format-control
                           :format-arguments format-arguments))))
   nil)
```

∗debugger-hook∗ *Variable*

Value Type:

a *designator* for a *function* of two *arguments* (a *condition* and the *value* of ***debugger-hook*** at the time the debugger was entered), or **nil**.

Initial Value:

nil.

Description:

When the *value* of ***debugger-hook*** is *non-nil*, it is called prior to normal entry into the debugger, either due to a call to **invoke-debugger** or due to automatic entry into the debugger from a call to **error** or **cerror** with a condition that is not handled. The *function* may either handle the *condition* (transfer control) or return normally (allowing the standard debugger to run). To minimize recursive errors while debugging, ***debugger-hook*** is bound to **nil** by **invoke-debugger** prior to calling the *function*.

Examples:

```
 (defun one-of (choices &optional (prompt "Choice"))
   (let ((n (length choices)) (i))
     (do ((c choices (cdr c)) (i 1 (+ i 1)))
         ((null c))
       (format t "~&[~D] ~A~%" i (car c)))
     (do () ((typep i `(integer 1 ,n)))
       (format t "~&~A: " prompt)
       (setq i (read))
```

```
       (fresh-line))
     (nth (- i 1) choices)))

 (defun my-debugger (condition me-or-my-encapsulation)
   (format t "~&Fooey: ~A" condition)
   (let ((restart (one-of (compute-restarts))))
     (if (not restart) (error "My debugger got an error."))
     (let ((*debugger-hook* me-or-my-encapsulation))
       (invoke-restart-interactively restart))))

 (let ((*debugger-hook* #'my-debugger))
   (+ 3 'a))
▷ Fooey: The argument to +, A, is not a number.
▷  [1] Supply a replacement for A.
▷  [2] Return to Cloe Toplevel.
▷ Choice: 1
▷  Form to evaluate and use: (+ 5 'b)
▷  Fooey: The argument to +, B, is not a number.
▷  [1] Supply a replacement for B.
▷  [2] Supply a replacement for A.
▷  [3] Return to Cloe Toplevel.
▷ Choice: 1
▷  Form to evaluate and use: 1
→ 9
```

Affected By:

invoke-debugger

Notes:

When evaluating code typed in by the user interactively, it is sometimes useful to have the hook function bind ***debugger-hook*** to the *function* that was its second argument so that recursive errors can be handled using the same interactive facility.

∗break-on-signals∗ *Variable*

Value Type:

a *type specifier*.

Initial Value:

nil.

Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

Description:

When (typep *condition* *break-on-signals*) returns *true*, calls to **signal**, and to other *operators* such as **error** that implicitly call **signal**, enter the debugger prior to *signaling* the *condition*.

The **continue** *restart* can be used to continue with the normal *signaling* process when a break occurs process due to ***break-on-signals***.

Examples:

```
 *break-on-signals* → NIL
 (ignore-errors (error 'simple-error :format-control "Fooey!"))
→ NIL, #<SIMPLE-ERROR 32207172>

 (let ((*break-on-signals* 'error))
   (ignore-errors (error 'simple-error :format-control "Fooey!")))
▷ Break: Fooey!
▷ BREAK entered because of *BREAK-ON-SIGNALS*.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Continue to signal.
▷  2: Top level.
▷ Debug> :CONTINUE 1
▷ Continue to signal.
→ NIL, #<SIMPLE-ERROR 32212257>

 (let ((*break-on-signals* 'error))
   (error 'simple-error :format-control "Fooey!"))
▷ Break: Fooey!
▷ BREAK entered because of *BREAK-ON-SIGNALS*.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Continue to signal.
▷  2: Top level.
▷ Debug> :CONTINUE 1
▷ Continue to signal.
▷ Error: Fooey!
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Top level.
▷ Debug> :CONTINUE 1
▷ Top level.
```

See Also:

break, **signal**, **warn**, **error**, **typep**, Section 9.1 (Condition System Concepts)

Notes:

break-on-signals is intended primarily for use in debugging code that does signaling. When setting ***break-on-signals***, the user is encouraged to choose the most restrictive specification that suffices. Setting ***break-on-signals*** effectively violates the modular handling of *condition* signaling. In practice, the complete effect of setting ***break-on-signals*** might be unpredictable in some cases since the user might not be aware of the variety or number of calls to **signal** that are used in code called only incidentally.

break-on-signals enables an early entry to the debugger but such an entry does not preclude an additional entry to the debugger in the case of operations such as **error** and **cerror**.

handler-bind *Macro*

Syntax:

handler-bind ({↓*binding*}*) {*form*}* → {*result*}*

binding::=(*type handler*)

Arguments and Values:

type—a *type specifier*.

handler—a *form*; evaluated to produce a *handler-function*.

handler-function—a *designator* for a *function* of one *argument*.

forms—an *implicit progn*.

results—the *values* returned by the *forms*.

Description:

Executes *forms* in a *dynamic environment* where the indicated *handler bindings* are in effect.

Each *handler* should evaluate to a *handler-function*, which is used to handle *conditions* of the given *type* during execution of the *forms*. This *function* should take a single argument, the *condition* being signaled.

If more than one *handler binding* is supplied, the *handler bindings* are searched sequentially from top to bottom in search of a match (by visual analogy with **typecase**). If an appropriate *type* is found, the associated handler is run in a *dynamic environment* where none of these *handler* bindings are visible (to avoid recursive errors). If the *handler declines*, the search continues for another *handler*.

If no appropriate *handler* is found, other *handlers* are sought from dynamically enclosing contours. If no *handler* is found outside, then **signal** returns or **error** enters the debugger.

Examples:

In the following code, if an unbound variable error is signaled in the body (and not handled by an intervening handler), the first function is called.


```
 (handler-bind ((unbound-variable #'(lambda ...))
                (error #'(lambda ...)))
   ...)
```

If any other kind of error is signaled, the second function is called. In either case, neither handler is active while executing the code in the associated function.

```
 (defun trap-error-handler (condition)
   (format *error-output* "~&~A~&" condition)
   (throw 'trap-errors nil))

 (defmacro trap-errors (&rest forms)
   `(catch 'trap-errors
      (handler-bind ((error #'trap-error-handler))
        ,@forms)))

 (list (trap-errors (signal "Foo.") 1)
       (trap-errors (error  "Bar.") 2)
       (+ 1 2))
▷ Bar.
→ (1 NIL 3)
```

Note that "Foo." is not printed because the condition made by **signal** is a *simple condition*, which is not of *type* **error**, so it doesn't trigger the handler for **error** set up by `trap-errors`.

See Also:

handler-case

handler-case *Macro*

Syntax:

handler-case *expression* ⟦{↓*error-clause*}* | ↓*no-error-clause*⟧ → {*result*}*

clause::=↓*error-clause* | ↓*no-error-clause*

error-clause::=(*typespec* ([*var*]) {*declaration*}* {*form*}*)

no-error-clause::=(`:no-error` *lambda-list* {*declaration*}* {*form*}*)

Arguments and Values:

expression—a *form*.

typespec—a *type specifier*.

var—a *variable name*.

lambda-list—an *ordinary lambda list*.

declaration—a **declare** *expression*; not evaluated.

form—a *form*.

results—In the normal situation, the values returned are those that result from the evaluation of *expression*; in the exceptional situation when control is transferred to a *clause*, the value of the last *form* in that *clause* is returned.

Description:

handler-case executes *expression* in a *dynamic environment* where various handlers are active. Each *error-clause* specifies how to handle a *condition* matching the indicated *typespec*. A *no-error-clause* allows the specification of a particular action if control returns normally.

If a *condition* is signaled for which there is an appropriate *error-clause* during the execution of *expression* (*i.e.*, one for which (typep *condition* '*typespec*) returns *true*) and if there is no intervening handler for a *condition* of that *type*, then control is transferred to the body of the relevant *error-clause*. In this case, the dynamic state is unwound appropriately (so that the handlers established around the *expression* are no longer active), and *var* is bound to the *condition* that had been signaled. If more than one case is provided, those cases are made accessible in parallel. That is, in

```
(handler-case form
  (typespec1 (var1) form1)
  (typespec2 (var2) form2))
```

if the first *clause* (containing *form1*) has been selected, the handler for the second is no longer visible (or vice versa).

The *clauses* are searched sequentially from top to bottom. If there is *type* overlap between *typespecs*, the earlier of the *clauses* is selected.

If *var* is not needed, it can be omitted. That is, a *clause* such as:

```
(typespec (var) (declare (ignore var)) form)
```

can be written (*typespec* () *form*).

If there are no *forms* in a selected *clause*, the case, and therefore **handler-case**, returns **nil**. If execution of *expression* returns normally and no *no-error-clause* exists, the values returned by *expression* are returned by **handler-case**. If execution of *expression* returns normally and a *no-error-clause* does exist, the values returned are used as arguments to the function described by constructing (lambda *lambda-list* {*form*}*) from the *no-error-clause*, and the *values* of that function call are returned by **handler-case**. The handlers which were established around the *expression* are no longer active at the time of this call.

Examples:

```
(defun assess-condition (condition)
  (handler-case (signal condition)
    (warning () "Lots of smoke, but no fire.")
    ((or arithmetic-error control-error cell-error stream-error)
       (condition)
      (format nil "~S looks especially bad." condition))
    (serious-condition (condition)
      (format nil "~S looks serious." condition))
    (condition () "Hardly worth mentioning.")))
→ ASSESS-CONDITION
(assess-condition (make-condition 'stream-error :stream *terminal-io*))
→ "#<STREAM-ERROR 12352256> looks especially bad."
(define-condition random-condition (condition) ()
  (:report (lambda (condition stream)
             (declare (ignore condition))
             (princ "Yow" stream))))
→ RANDOM-CONDITION
(assess-condition (make-condition 'random-condition))
→ "Hardly worth mentioning."
```

See Also:

handler-bind, **ignore-errors**, Section 9.1 (Condition System Concepts)

Notes:

```
(handler-case form
  (type1 (var1) . body1)
  (type2 (var2) . body2) ...)
```

is approximately equivalent to:

```
(block #1=#:g0001
  (let ((#2=#:g0002 nil))
    (tagbody
      (handler-bind ((type1 #'(lambda (temp)
                                (setq #1# temp)
                                (go #3=#:g0003)))
                     (type2 #'(lambda (temp)
                                (setq #2# temp)
                                (go #4=#:g0004))) ...)
      (return-from #1# form))
        #3# (return-from #1# (let ((var1 #2#)) . body1))
        #4# (return-from #1# (let ((var2 #2#)) . body2)) ...)))
```

```
(handler-case form
```

```
(type1 (var1) . body1)
...
(:no-error (varN-1 varN-2 ...) . bodyN))
```

is approximately equivalent to:

```
(block #1=#:error-return
 (multiple-value-call #'(lambda (varN-1 varN-2 ...) . bodyN)
    (block #2=#:normal-return
      (return-from #1#
        (handler-case (return-from #2# form)
          (type1 (var1) . body1) ...)))))
```

ignore-errors *Macro*

Syntax:

ignore-errors {*form*}* → {*result*}*

Arguments and Values:

forms—an *implicit progn*.

results—In the normal situation, the *values* of the *forms* are returned; in the exceptional situation, two values are returned: **nil** and the *condition*.

Description:

ignore-errors is used to prevent *conditions* of *type* **error** from causing entry into the debugger.

Specifically, **ignore-errors** *executes forms* in a *dynamic environment* where a *handler* for *conditions* of *type* **error** has been established; if invoked, it *handles* such *conditions* by returning two *values*, **nil** and the *condition* that was *signaled*, from the **ignore-errors** *form*.

If a *normal return* from the *forms* occurs, any *values* returned are returned by **ignore-errors**.

Examples:

```
(defun load-init-file (program)
  (let ((win nil))
    (ignore-errors ;if this fails, don't enter debugger
      (load (merge-pathnames (make-pathname :name program :type :lisp)
                             (user-homedir-pathname)))
      (setq win t))
    (unless win (format t "~&Init file failed to load.~%"))
    win))
```


Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

```
(load-init-file "no-such-program")
▷ Init file failed to load.
NIL
```

See Also:

handler-case, Section 9.1 (Condition System Concepts)

Notes:

```
(ignore-errors . forms)
```

is equivalent to:

```
(handler-case (progn . forms)
  (error (condition) (values nil condition)))
```

Because the second return value is a *condition* in the exceptional case, it is common (but not required) to arrange for the second return value in the normal case to be missing or **nil** so that the two situations can be distinguished.

define-condition *Macro*

Syntax:

define-condition *name* ({*parent-type*}*) ({↓*slot-spec*}*) {*option*}*
→ *name*

slot-spec::=*slot-name* | (*slot-name* ↓*slot-option*)

slot-option::=⟦{:reader *symbol*}* |
{:writer ↓*function-name*}* |
{:accessor *symbol*}* |
{:allocation ↓*allocation-type*} |
{:initarg *symbol*}* |
{:initform *form*} |
{:type *type-specifier*} ⟧

option::=⟦(:default-initargs . *initarg-list*) |
(:documentation *string*) |
(:report *report-name*) ⟧

function-name::={*symbol* | (setf *symbol*)}

allocation-type::=:instance | :class

report-name::=*string* | *symbol* | *lambda expression*

Arguments and Values:

name—a *symbol*.

parent-type—a *symbol* naming a *condition type*. If no *parent-types* are supplied, the *parent-types* default to (condition).

default-initargs—a *list* of *keyword/value pairs*.

Slot-spec – the *name* of a *slot* or a *list* consisting of the *slot-name* followed by zero or more *slot-options*.

Slot-name – a slot name (a *symbol*), the *list* of a slot name, or the *list* of slot name/slot form pairs.

Option – Any of the following:

:reader

:reader can be supplied more than once for a given *slot* and cannot be **nil**.

:writer

:writer can be supplied more than once for a given *slot* and must name a *generic function*.

:accessor

:accessor can be supplied more than once for a given *slot* and cannot be **nil**.

:allocation

:allocation can be supplied once at most for a given *slot*. The default if :allocation is not supplied is :instance.

:initarg

:initarg can be supplied more than once for a given *slot*.

:initform

:initform can be supplied once at most for a given *slot*.

:type

:type can be supplied once at most for a given *slot*.

:documentation

:documentation can be supplied once at most for a given *slot*.

:report

:report can be supplied once at most.

Description:

define-condition defines a new condition type called *name*, which is a *subtype* of the *type* or *types* named by *parent-type*. Each *parent-type* argument specifies a direct *supertype* of the new *condition*. The new *condition* inherits *slots* and *methods* from each of its direct *supertypes*, and so on.

If a slot name/slot form pair is supplied, the slot form is a *form* that can be evaluated by **make-condition** to produce a default value when an explicit value is not provided. If no slot form is supplied, the contents of the *slot* is initialized in an *implementation-dependent* way.

If the *type* being defined and some other *type* from which it inherits have a slot by the same name, only one slot is allocated in the *condition*, but the supplied slot form overrides any slot form that might otherwise have been inherited from a *parent-type*. If no slot form is supplied, the inherited slot form (if any) is still visible.

Accessors are created according to the same rules as used by **defclass**.

A description of *slot-options* follows:

:reader

The :reader slot option specifies that an *unqualified method* is to be defined on the *generic function* named by the argument to :reader to read the value of the given *slot*.

- The :initform slot option is used to provide a default initial value form to be used in the initialization of the *slot*. This *form* is evaluated every time it is used to initialize the *slot*. The *lexical environment* in which this *form* is evaluated is the lexical *environment* in which the **define-condition** form was evaluated. Note that the *lexical environment* refers both to variables and to *functions*. For *local slots*, the *dynamic environment* is the dynamic *environment* in which **make-condition** was called; for *shared slots*, the *dynamic environment* is the *dynamic environment* in which the **define-condition** form was evaluated.

  No implementation is permitted to extend the syntax of **define-condition** to allow (*slot-name form*) as an abbreviation for (*slot-name* :initform *form*).

:initarg

The :initarg slot option declares an initialization argument named by its *symbol* argument and specifies that this initialization argument initializes the given *slot*. If the initialization argument has a value in the call to **initialize-instance**, the value is stored into the given *slot*, and the slot's `:initform` slot option, if any, is not evaluated. If none of the initialization arguments specified for a given *slot* has a value, the *slot* is initialized according to the `:initform` slot option, if specified.

`:type`

The `:type` slot option specifies that the contents of the *slot* is always of the specified *type*. It effectively declares the result type of the reader generic function when applied to an *object* of this *condition* type. The consequences of attempting to store in a *slot* a value that does not satisfy the type of the *slot* is undefined.

`:default-initargs`

This option is treated the same as it would be **defclass**.

`:documentation`

The `:documentation` slot option provides a *documentation string* for the *slot*.

`:report`

Condition reporting is mediated through the **print-object** method for the *condition* type in question, with ***print-escape*** always being **nil**. Specifying `(:report report-name)` in the definition of a condition type `C` is equivalent to:

```
(defmethod print-object ((x c) stream)
  (if *print-escape* (call-next-method) (report-name x stream)))
```

If the value supplied by the argument to `:report` (*report-name*) is a *symbol* or a *lambda expression*, it must be acceptable to **function**. `(function report-name)` is evaluated in the current *lexical environment*. It should return a *function* of two arguments, a *condition* and a *stream*, that prints on the *stream* a description of the *condition*. This *function* is called whenever the *condition* is printed while ***print-escape*** is **nil**.

If *report-name* is a *string*, it is a shorthand for

```
(lambda (condition stream)
  (declare (ignore condition))
  (write-string report-name stream))
```

This option is processed after the new *condition* type has been defined, so use of the *slot* accessors within the `:report` function is permitted. If this option is not supplied, information about how to report this type of *condition* is inherited from the *parent-type*.

The consequences are unspecifed if an attempt is made to *read* a *slot* that has not been explicitly initialized and that has not been given a default value.

The consequences are unspecified if an attempt is made to assign the *slots* by using **setf**.

If a **define-condition** *form* appears as a *top level form*, the *compiler* must make *name* recognizable as a valid *type* name, and it must be possible to reference the *condition type* as the *parent-type* of another *condition type* in a subsequent **define-condition** *form* in the *file* being compiled.

Examples:

The following form defines a condition of *type* `peg/hole-mismatch` which inherits from a condition type called `blocks-world-error`:

```
(define-condition peg/hole-mismatch
                  (blocks-world-error)
                  ((peg-shape  :initarg :peg-shape
                               :reader peg/hole-mismatch-peg-shape)
                   (hole-shape :initarg :hole-shape
                               :reader peg/hole-mismatch-hole-shape))
  (:report (lambda (condition stream)
             (format stream "A ~A peg cannot go in a ~A hole."
                     (peg/hole-mismatch-peg-shape  condition)
                     (peg/hole-mismatch-hole-shape condition)))))
```

The new type has slots `peg-shape` and `hole-shape`, so **make-condition** accepts `:peg-shape` and `:hole-shape` keywords. The *readers* `peg/hole-mismatch-peg-shape` and `peg/hole-mismatch-hole-shape` apply to objects of this type, as illustrated in the `:report` information.

The following form defines a *condition type* named `machine-error` which inherits from **error**:

```
(define-condition machine-error
                  (error)
                  ((machine-name :initarg :machine-name
                                 :reader machine-error-machine-name))
  (:report (lambda (condition stream)
             (format stream "There is a problem with ~A."
                     (machine-error-machine-name condition)))))
```

Building on this definition, a new error condition can be defined which is a subtype of `machine-error` for use when machines are not available:

```
(define-condition machine-not-available-error (machine-error) ()
  (:report (lambda (condition stream)
             (format stream "The machine ~A is not available."
                     (machine-error-machine-name condition)))))
```

This defines a still more specific condition, built upon `machine-not-available-error`, which provides a slot initialization form for `machine-name` but which does not provide any new slots or report information. It just gives the `machine-name` slot a default initialization:

```
(define-condition my-favorite-machine-not-available-error
                  (machine-not-available-error)
  ((machine-name :initform "mc.lcs.mit.edu")))
```

Note that since no `:report` clause was given, the information inherited from `machine-not-available-error` is used to report this type of condition.

```
 (define-condition ate-too-much (error)
     ((person :initarg :person :reader ate-too-much-person)
      (weight :initarg :weight :reader ate-too-much-weight)
      (kind-of-food :initarg :kind-of-food
                    :reader :ate-too-much-kind-of-food)))
→ ATE-TOO-MUCH
 (define-condition ate-too-much-ice-cream (ate-too-much)
   ((kind-of-food :initform 'ice-cream)
    (flavor       :initarg :flavor
                  :reader ate-too-much-ice-cream-flavor
                  :initform 'vanilla ))
   (:report (lambda (condition stream)
              (format stream "~A ate too much ~A ice-cream"
                      (ate-too-much-person condition)
                      (ate-too-much-ice-cream-flavor condition)))))
→ ATE-TOO-MUCH-ICE-CREAM
 (make-condition 'ate-too-much-ice-cream
                 :person 'fred
                 :weight 300
                 :flavor 'chocolate)
→ #<ATE-TOO-MUCH-ICE-CREAM 32236101>
 (format t "~A" *)
▷ FRED ate too much CHOCOLATE ice-cream
→ NIL
```

See Also:

make-condition, **defclass**, Section 9.1 (Condition System Concepts)

make-condition *Function*

Syntax:

make-condition *type* &rest *slot-initializations* → *condition*

Arguments and Values:

type—a *type specifier* (for a *subtype* of **condition**).

slot-initializations—an *initialization argument list*.

condition—a *condition*.

Description:

Constructs and returns a *condition* of type *type* using *slot-initializations* for the initial values of the slots. The newly created *condition* is returned.

Examples:

```
 (defvar *oops-count* 0)

 (setq a (make-condition 'simple-error
                         :format-control "This is your ~:R error."
                         :format-arguments (list (incf *oops-count*))))
→ #<SIMPLE-ERROR 32245104>

 (format t "~&~A~%" a)
▷ This is your first error.
→ NIL

 (error a)
▷ Error: This is your first error.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Return to Lisp Toplevel.
▷ Debug>
```

Affected By:

The set of defined *condition types*.

See Also:

define-condition, Section 9.1 (Condition System Concepts)

restart *System Class*

Class Precedence List:

restart, **t**

Description:

An *object* of *type* **restart** represents a *function* that can be called to perform some form of recovery action, usually a transfer of control to an outer point in the running program.

An *implementation* is free to implement a *restart* in whatever manner is most convenient; a *restart* has only *dynamic extent* relative to the scope of the binding *form* which *establishes* it.

compute-restarts *Function*

Syntax:

compute-restarts &optional *condition* → *restarts*

Arguments and Values:

condition—a *condition object*, or **nil**.

restarts—a *list* of *restarts*.

Description:

compute-restarts uses the dynamic state of the program to compute a *list* of the *restarts* which are currently active.

The resulting *list* is ordered so that the innermost (more-recently established) restarts are nearer the head of the *list*.

When *condition* is *non-nil*, only those *restarts* are considered that are either explicitly associated with that *condition*, or not associated with any *condition*; that is, the excluded *restarts* are those that are associated with a non-empty set of *conditions* of which the given *condition* is not an *element*. If *condition* is **nil**, all *restarts* are considered.

compute-restarts returns all *applicable restarts*, including anonymous ones, even if some of them have the same name as others and would therefore not be found by **find-restart** when given a *symbol* argument.

Implementations are permitted, but not required, to return *distinct lists* from repeated calls to **compute-restarts** while in the same dynamic environment. The consequences are undefined if the *list* returned by **compute-restarts** is every modified.

Examples:

```
;; One possible way in which an interactive debugger might present
;; restarts to the user.
(defun invoke-a-restart ()
  (let ((restarts (compute-restarts)))
    (do ((i 0 (+ i 1)) (r restarts (cdr r))) ((null r))
      (format t "~&~D: ~A~%" i (car r)))
    (let ((n nil) (k (length restarts)))
      (loop (when (and (typep n 'integer) (>= n 0) (< n k))
```



```
               (return t))
             (format t "~&Option: ")
             (setq n (read))
             (fresh-line))
          (invoke-restart-interactively (nth n restarts)))))

 (restart-case (invoke-a-restart)
   (one () 1)
   (two () 2)
   (nil () :report "Who knows?" 'anonymous)
   (one () 'I)
   (two () 'II))
▷ 0: ONE
▷ 1: TWO
▷ 2: Who knows?
▷ 3: ONE
▷ 4: TWO
▷ 5: Return to Lisp Toplevel.
▷ Option: 4
→ II

 ;; Note that in addition to user-defined restart points, COMPUTE-RESTARTS
 ;; also returns information about any system-supplied restarts, such as
 ;; the "Return to Lisp Toplevel" restart offered above.
```

Affected By:

Existing restarts.

See Also:

find-restart, **invoke-restart**, **restart-bind**

find-restart *Function*

Syntax:

find-restart *identifier* &optional *condition*

restart

Arguments and Values:

identifier—a *non-nil symbol*, or a *restart*.

condition—a *condition object*, or **nil**.

restart—a *restart* or **nil**.

Description:

find-restart searches for a particular *restart* in the current *dynamic environment*.

When *condition* is *non-nil*, only those *restarts* are considered that are either explicitly associated with that *condition*, or not associated with any *condition*; that is, the excluded *restarts* are those that are associated with a non-empty set of *conditions* of which the given *condition* is not an *element*. If *condition* is **nil**, all *restarts* are considered.

If *identifier* is a *symbol*, then the innermost (most recently established) *applicable restart* with that *name* is returned. **nil** is returned if no such restart is found.

If *identifier* is a currently active restart, then it is returned. Otherwise, **nil** is returned.

Examples:

```
 (restart-case
     (let ((r (find-restart 'my-restart)))
       (format t "~S is named ~S" r (restart-name r)))
   (my-restart () nil))
▷ #<RESTART 32307325> is named MY-RESTART
→ NIL
 (find-restart 'my-restart)
→ NIL
```

Affected By:

Existing restarts.

restart-case, **restart-bind**, **with-condition-restarts**.

See Also:

compute-restarts

Notes:

```
 (find-restart identifier)
 ≡ (find identifier (compute-restarts) :key :restart-name)
```

Although anonymous restarts have a name of **nil**, the consequences are unspecified if **nil** is given as an *identifier*. Occasionally, programmers lament that **nil** is not permissible as an *identifier* argument. In most such cases, **compute-restarts** can probably be used to simulate the desired effect.

invoke-restart *Function*

Syntax:

invoke-restart *restart* &rest *arguments* → {*result*}*

Arguments and Values:

restart—a *restart designator*.

argument—an *object*.

results—the *values* returned by the *function* associated with *restart*, if that *function* returns.

Description:

Calls the *function* associated with *restart*, passing *arguments* to it. *Restart* must be valid in the current *dynamic environment*.

Examples:

```
 (defun add3 (x) (check-type x number) (+ x 3))

 (foo 'seven)
▷ Error: The value SEVEN was not of type NUMBER.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a different value to use.
▷  2: Return to Lisp Toplevel.
▷ Debug> (invoke-restart 'store-value 7)
→ 10
```

Side Effects:

A non-local transfer of control might be done by the restart.

Affected By:

Existing restarts.

Exceptional Situations:

If *restart* is not valid, an error of *type* **control-error** is signaled.

See Also:

find-restart, **restart-bind**, **restart-case**, **invoke-restart-interactively**

Notes:

The most common use for **invoke-restart** is in a *handler*. It might be used explicitly, or implicitly through **invoke-restart-interactively** or a *restart function*.

Restart functions call **invoke-restart**, not vice versa. That is, *invoke-restart* provides primitive functionality, and *restart functions* are non-essential "syntactic sugar."

invoke-restart-interactively *Function*

Syntax:

invoke-restart-interactively *restart* → {*result*}*

Arguments and Values:

restart—a *restart designator*.

results—the *values* returned by the *function* associated with *restart*, if that *function* returns.

Description:

invoke-restart-interactively calls the *function* associated with *restart*, prompting for any necessary arguments. If *restart* is a name, it must be valid in the current *dynamic environment*.

invoke-restart-interactively prompts for arguments by executing the code provided in the `:interactive` keyword to **restart-case** or `:interactive-function` keyword to **restart-bind**.

If no such options have been supplied in the corresponding **restart-bind** or **restart-case**, then the consequences are undefined if the *restart* takes required arguments. If the arguments are optional, an argument list of **nil** is used.

Once the arguments have been determined, **invoke-restart-interactively** executes the following:

```
(apply #'invoke-restart restart arguments)
```

Examples:

```
(defun add3 (x) (check-type x number) (+ x 3))

(add3 'seven)
▷ Error: The value SEVEN was not of type NUMBER.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a different value to use.
▷  2: Return to Lisp Toplevel.
▷ Debug> (invoke-restart-interactively 'store-value)
▷ Type a form to evaluate and use: 7
→ 10
```

Side Effects:

If prompting for arguments is necesary, some typeout may occur (on *query I/O*).

A non-local transfer of control might be done by the restart.

Affected By:

query-io, active *restarts*

Exceptional Situations:

If *restart* is not valid, an error of *type* **control-error** is signaled.

See Also:

find-restart, **invoke-restart**, **restart-case**, **restart-bind**

Notes:

invoke-restart-interactively is used internally by the debugger and may also be useful in implementing other portable, interactive debugging tools.

restart-bind *Macro*

Syntax:

restart-bind ({(*name function* {↓*key-val-pair*}*)}) {*form*}*
→ {*result*}*

key-val-pair::= `:interactive-function` *interactive-function* |
`:report-function` *report-function* |
`:test-function` *test-function*

Arguments and Values:

name—a *symbol*; not evaluated.

function—a *form*; evaluated.

forms—an *implicit progn*.

interactive-function—a *form*; evaluated.

report-function—a *form*; evaluated.

test-function—a *form*; evaluated.

results—the *values* returned by the *forms*.

Description:

restart-bind executes the body of *forms* in a *dynamic environment* where *restarts* with the given *names* are in effect.

If a *name* is **nil**, it indicates an anonymous restart; if a *name* is a *non-nil symbol*, it indicates a named restart.

The *function*, *interactive-function*, and *report-function* are unconditionally evaluated in the current lexical and dynamic environment prior to evaluation of the body. Each of these *forms* must evaluate to a *function*.

If **invoke-restart** is done on that restart, the *function* which resulted from evaluating *function* is called, in the *dynamic environment* of the **invoke-restart**, with the *arguments* given to **invoke-restart**. The *function* may either perform a non-local transfer of control or may return normally.

If the restart is invoked interactively from the debugger (using **invoke-restart-interactively**), the arguments are defaulted by calling the *function* which resulted from evaluating *interactive-function*. That *function* may optionally prompt interactively on *query I/O*, and should return a *list* of arguments to be used by **invoke-restart-interactively** when invoking the restart.

If a restart is invoked interactively but no *interactive-function* is used, then an argument list of **nil** is used. In that case, the *function* must be compatible with an empty argument list.

If the restart is presented interactively (*e.g.*, by the debugger), the presentation is done by calling the *function* which resulted from evaluating *report-function*. This *function* must be a *function* of one argument, a *stream*. It is expected to print a description of the action that the restart takes to that *stream*. This *function* is called any time the restart is printed while ***print-escape*** is **nil**.

In the case of interactive invocation, the result is dependent on the value of `:interactive-function` as follows.

`:interactive-function`

> *Value* is evaluated in the current lexical environment and should return a *function* of no arguments which constructs a *list* of arguments to be used by **invoke-restart-interactively** when invoking this restart. The *function* may prompt interactively using *query I/O* if necessary.

`:report-function`

> *Value* is evaluated in the current lexical environment and should return a *function* of one argument, a *stream*, which prints on the *stream* a summary of the action that this restart takes. This *function* is called whenever the restart is reported (printed while ***print-escape*** is **nil**). If no `:report-function` option is provided, the manner in which the *restart* is reported is *implementation-dependent*.

`:test-function`

> *Value* is evaluated in the current lexical environment and should return a *function* of one argument, a *condition*, which returns *true* if the restart is to be considered visible.

Affected By:

query-io.

See Also:

restart-case, **with-simple-restart**

Notes:

restart-bind is primarily intended to be used to implement **restart-case** and might be useful in implementing other macros. Programmers who are uncertain about whether to use **restart-case** or **restart-bind** should prefer **restart-case** for the cases where it is powerful enough, using **restart-bind** only in cases where its full generality is really needed.

restart-case *Macro*

Syntax:

restart-case *restartable-form* {↓*clause*} → {*result*}*

clause::=(*case-name lambda-list*
⟦`:interactive` *interactive-expression* | `:report` *report-expression* | `:test` *test-expression*⟧
{*declaration*}* {*form*}*)

Arguments and Values:

restartable-form—a *form*.

case-name—a *symbol* or **nil**.

lambda-list—an *ordinary lambda list*.

interactive-expression—a *symbol* or a *lambda expression*.

report-expression—a *string*, a *symbol*, or a *lambda expression*.

test-expression—a *symbol* or a *lambda expression*.

declaration—a **declare** *expression*; not evaluated.

form—a *form*.

results—the *values* resulting from the *evaluation* of *restartable-form*, or the *values* returned by the last *form* executed in a chosen *clause*, or **nil**.

restart-case

Description:

restart-case evaluates *restartable-form* in a *dynamic environment* where the clauses have special meanings as points to which control may be transferred. If *restartable-form* finishes executing and returns any values, all values returned are returned by **restart-case** and processing has completed. While *restartable-form* is executing, any code may transfer control to one of the clauses (see **invoke-restart**). If a transfer occurs, the forms in the body of that clause is evaluated and any values returned by the last such form are returned by **restart-case**. In this case, the dynamic state is unwound appropriately (so that the restarts established around the *restartable-form* are no longer active) prior to execution of the clause.

If there are no *forms* in a selected clause, **restart-case** returns **nil**.

If *case-name* is a *symbol*, it names this restart.

It is possible to have more than one clause use the same *case-name*. In this case, the first clause with that name is found by **find-restart**. The other clauses are accessible using **compute-restarts**.

Each *arglist* is an *ordinary lambda list* to be bound during the execution of its corresponding *forms*. These parameters are used by the **restart-case** clause to receive any necessary data from a call to **invoke-restart**.

By default, **invoke-restart-interactively** passes no arguments and all arguments must be optional in order to accomodate interactive restarting. However, the arguments need not be optional if the `:interactive` keyword has been used to inform **invoke-restart-interactively** about how to compute a proper argument list.

Keyword options have the following meaning.

`:interactive`

The *value* supplied by `:interactive` *value* must be a suitable argument to **function**. `(function` *value*`)` is evaluated in the current lexical environment. It should return a *function* of no arguments which returns arguments to be used by **invoke-restart-interactively** when it is invoked. **invoke-restart-interactively** is called in the dynamic environment available prior to any restart attempt, and uses *query I/O* for user interaction.

If a restart is invoked interactively but no `:interactive` option was supplied, the argument list used in the invocation is the empty list.

`:report`

If the *value* supplied by `:report` *value* is a *lambda expression* or a *symbol*, it must be acceptable to **function**. `(function` *value*`)` is evaluated in the current lexical environment. It should return a *function* of one argument, a *stream*, which prints on the *stream* a description of the restart. This *function* is called whenever the restart is printed while ***print-escape*** is **nil**.

If *value* is a *string*, it is a shorthand for

```
(lambda (stream) (write-string value stream))
```

If a named restart is asked to report but no report information has been supplied, the name of the restart is used in generating default report text.

When ***print-escape*** is **nil**, the printer uses the report information for a restart. For example, a debugger might announce the action of typing a "continue" command by:

```
(format t "~&~S -- ~A~%" ':continue some-restart)
```

which might then display as something like:

```
:CONTINUE -- Return to command level
```

The consequences are unspecified if an unnamed restart is specified but no `:report` option is provided.

`:test`

The *value* supplied by `:test` *value* must be a suitable argument to **function**. `(function` *value*`)` is evaluated in the current lexical environment. It should return a *function* of one *argument*, the *condition*, that returns *true* if the restart is to be considered visible.

The default for this option is equivalent to `(lambda (c) (declare (ignore c)) t)`.

If the *restartable-form* is a *list* whose *car* is any of the *symbols* **signal**, **error**, **cerror**, or **warn** (or is a *macro form* which macroexpands into such a *list*), then **with-condition-restarts** is used implicitly to associate the indicated *restarts* with the *condition* to be signaled.

Examples:

```
(restart-case
    (handler-bind ((error #'(lambda (c)
                              (declare (ignore condition))
                              (invoke-restart 'my-restart 7))))
      (error "Foo."))
  (my-restart (&optional v) v))
→ 7

(define-condition food-error (error) ())
→ FOOD-ERROR
(define-condition bad-tasting-sundae (food-error)
  ((ice-cream :initarg :ice-cream :reader bad-tasting-sundae-ice-cream)
   (sauce :initarg :sauce :reader bad-tasting-sundae-sauce)
   (topping :initarg :topping :reader bad-tasting-sundae-topping))
  (:report (lambda (condition stream)
```

```
             (format stream "Bad tasting sundae with ~S, ~S, and ~S"
                     (bad-tasting-sundae-ice-cream condition)
                     (bad-tasting-sundae-sauce condition)
                     (bad-tasting-sundae-topping condition)))))
→ BAD-TASTING-SUNDAE
 (defun all-start-with-same-letter (symbol1 symbol2 symbol3)
   (let ((first-letter (char (symbol-name symbol1) 0)))
     (and (eql first-letter (char (symbol-name symbol2) 0))
          (eql first-letter (char (symbol-name symbol3) 0)))))
→ ALL-START-WITH-SAME-LETTER
 (defun read-new-value ()
   (format t "Enter a new value: ")
   (multiple-value-list (eval (read))))
→ READ-NEW-VALUE
```



```
 (defun verify-or-fix-perfect-sundae (ice-cream sauce topping)
   (do ()
      ((all-start-with-same-letter ice-cream sauce topping))
     (restart-case
       (error 'bad-tasting-sundae
              :ice-cream ice-cream
              :sauce sauce
              :topping topping)
       (use-new-ice-cream (new-ice-cream)
         :report "Use a new ice cream."
         :interactive read-new-value
         (setq ice-cream new-ice-cream))
       (use-new-sauce (new-sauce)
         :report "Use a new sauce."
         :interactive read-new-value
         (setq sauce new-sauce))
       (use-new-topping (new-topping)
         :report "Use a new topping."
         :interactive read-new-value
         (setq topping new-topping))))
   (values ice-cream sauce topping))
→ VERIFY-OR-FIX-PERFECT-SUNDAE
 (verify-or-fix-perfect-sundae 'vanilla 'caramel 'cherry)
▷ Error: Bad tasting sundae with VANILLA, CARAMEL, and CHERRY.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Use a new ice cream.
▷  2: Use a new sauce.
▷  3: Use a new topping.
▷  4: Return to Lisp Toplevel.
▷ Debug> :continue 1
▷ Use a new ice cream.
▷ Enter a new ice cream: 'chocolate
→ CHOCOLATE, CARAMEL, CHERRY
```

See Also:

restart-bind, **with-simple-restart**.

Notes:

(restart-case *expression*
   (*name1 arglist1 ...options1... . body1*)
   (*name2 arglist2 ...options2... . body2*))

is essentially equivalent to

```
(block #1=#:g0001
  (let ((#2=#:g0002 nil))
       (tagbody
       (restart-bind ((name1 #'(lambda (&rest temp)
                                (setq #2# temp)
                                (go #3=#:g0003))
                          ...slightly-transformed-options1...)
                      (name2 #'(lambda (&rest temp)
                                (setq #2# temp)
                                (go #4=#:g0004))
                          ...slightly-transformed-options2...))
       (return-from #1# expression))
         #3# (return-from #1#
                 (apply #'(lambda arglist1 . body1) #2#))
         #4# (return-from #1#
                 (apply #'(lambda arglist2 . body2) #2#)))))
```

Unnamed restarts are generally only useful interactively and an interactive option which has no description is of little value. Implementations are encouraged to warn if an unnamed restart is used and no report information is provided at compilation time. At runtime, this error might be noticed when entering the debugger. Since signaling an error would probably cause recursive entry into the debugger (causing yet another recursive error, etc.) it is suggested that the debugger print some indication of such problems when they occur but not actually signal errors.

```
(restart-case (signal fred)
  (a ...)
  (b ...))
≡
(restart-case
    (with-condition-restarts fred
                             (list (find-restart 'a)
                                   (find-restart 'b))
      (signal fred))
  (a ...)
  (b ...))
```

restart-name *Function*

Syntax:

restart-name *restart* → *name*

Arguments and Values:

restart—a *restart*.

name—a *symbol*.

Description:

Returns the name of the *restart*, or **nil** if the *restart* is not named.

Examples:

```
(restart-case
    (loop for restart in (compute-restarts)
              collect (restart-name restart))
  (case1 () :report "Return 1." 1)
  (nil   () :report "Return 2." 2)
  (case3 () :report "Return 3." 3)
  (case1 () :report "Return 4." 4))
→ (CASE1 NIL CASE3 CASE1 ABORT)
;; In the example above the restart named ABORT was not created
;; explicitly, but was implicitly supplied by the system.
```

See Also:

compute-restarts, **find-restart**

with-condition-restarts *Macro*

Syntax:

with-condition-restarts *condition-form restarts-form* {*form*}*
→ {*result*}*

Arguments and Values:

condition-form—a *form*; *evaluated* to produce a *condition*.

condition—a *condition object* resulting from the *evaluation* of *condition-form*.

restart-form—a *form*; *evaluated* to produce a *restart-list*.

restart-list—a *list* of *restart objects* resulting from the *evaluation* of *restart-form*.

forms—an *implicit progn*; evaluated.

results—the *values* returned by *forms*.

Description:

First, the *condition-form* and *restarts-form* are *evaluated* in normal left-to-right order; the *primary values* yielded by these *evaluations* are respectively called the *condition* and the *restart-list*.

Next, the *forms* are *evaluated* in a *dynamic environment* in which each *restart* in *restart-list* is associated with the *condition*. See Section 9.1.4.2.4 (Associating a Restart with a Condition).

See Also:

restart-case

Notes:

Usually this *macro* is not used explicitly in code, since **restart-case** handles most of the common cases in a way that is syntactically more concise.

with-simple-restart *Macro*

Syntax:

with-simple-restart (*name format-control* {*format-argument*}*) {*form*}*
→ {*result*}*

Arguments and Values:

name—a *symbol*.

format-control—a *format control*.

format-argument—an *object* (*i.e.*, a *format argument*).

forms—an *implicit progn*.

results—in the normal situation, the *values* returned by the *forms*; in the exceptional situation where the *restart* named *name* is invoked, two values—**nil** and **t**.

Description:

with-simple-restart establishes a restart.

If the restart designated by *name* is not invoked while executing *forms*, all values returned by the last of *forms* are returned. If the restart designated by *name* is invoked, control is transferred to **with-simple-restart**, which returns two values, **nil** and **t**.

If *name* is **nil**, an anonymous restart is established.

The *format-control* and *format-arguments* are used report the *restart*.

Examples:

```
 (defun read-eval-print-loop (level)
   (with-simple-restart (abort "Exit command level ~D." level)
     (loop
       (with-simple-restart (abort "Return to command level ~D." level)
         (let ((form (prog2 (fresh-line) (read) (fresh-line))))
           (prin1 (eval form)))))))
→ READ-EVAL-PRINT-LOOP
 (read-eval-print-loop 1)
 (+ 'a 3)
▷ Error: The argument, A, to the function + was of the wrong type.
▷        The function expected a number.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use this time.
▷  2: Return to command level 1.
▷  3: Exit command level 1.
▷  4: Return to Lisp Toplevel.

 (defun compute-fixnum-power-of-2 (x)
   (with-simple-restart (nil "Give up on computing 2^~D." x)
     (let ((result 1))
       (dotimes (i x result)
         (setq result (* 2 result))
         (unless (fixnump result)
           (error "Power of 2 is too large."))))))
COMPUTE-FIXNUM-POWER-OF-2
 (defun compute-power-of-2 (x)
   (or (compute-fixnum-power-of-2 x) 'something big))
COMPUTE-POWER-OF-2
 (compute-power-of-2 10)
1024
 (compute-power-of-2 10000)
▷ Error: Power of 2 is too large.
▷ To continue, type :CONTINUE followed by an option number.
▷  1: Give up on computing 2^10000.
▷  2: Return to Lisp Toplevel
▷ Debug> :continue 1
→ SOMETHING-BIG
```

See Also:

restart-case

Notes:

with-simple-restart is shorthand for one of the most common uses of **restart-case**.

with-simple-restart could be defined by:

```
(defmacro with-simple-restart ((restart-name format-control
                                              &rest format-arguments)
                               &body forms)
  `(restart-case (progn ,@forms)
     (,restart-name ()
        :report (lambda (stream)
                  (format stream ,format-control ,@format-arguments))
       (values nil t))))
```

Because the second return value is **t** in the exceptional case, it is common (but not required) to arrange for the second return value in the normal case to be missing or **nil** so that the two situations can be distinguished.

abort *Restart*

Data Arguments Required:

None.

Description:

The intent of the **abort** restart is to allow return to the innermost "command level." Implementors are encouraged to make sure that there is always a restart named **abort** around any user code so that user code can call **abort** at any time and expect something reasonable to happen; exactly what the reasonable thing is may vary somewhat. Typically, in an interactive listener, the invocation of **abort** returns to the *Lisp reader* phase of the *Lisp read-eval-print loop*, though in some batch or multi-processing situations there may be situations in which having it kill the running process is more appropriate.

See Also:

Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), **invoke-restart**, **abort** (*function*)

continue *Restart*

Data Arguments Required:

None.

Description:

The **continue** *restart* is generally part of protocols where there is a single "obvious" way to continue, such as in **break** and **cerror**. Some user-defined protocols may also wish to incorporate it for similar reasons. In general, however, it is more reliable to design a special purpose restart with a name that more directly suits the particular application.

Examples:

```
(let ((x 3))
  (handler-bind ((error #'(lambda (c)
                           (let ((r (find-restart 'continue c)))
                             (when r (invoke-restart r))))))
    (cond ((not (floatp x))
           (cerror "Try floating it." "~D is not a float." x)
           (float x))
          (t x)))) → 3.0
```

See Also:

Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), **invoke-restart**, **continue** (*function*), **assert**, **cerror**

muffle-warning *Restart*

Data Arguments Required:

None.

Description:

This *restart* is established by **warn** so that *handlers* of **warning** *conditions* have a way to tell **warn** that a warning has already been dealt with and that no further action is warranted.

Examples:

```
(defvar *all-quiet* nil) → *ALL-QUIET*
(defvar *saved-warnings* '()) → *SAVED-WARNINGS*
(defun quiet-warning-handler (c)
  (when *all-quiet*
    (let ((r (find-restart 'muffle-warning c)))
```

```
      (when r
        (push c *saved-warnings*)
        (invoke-restart r)))))
→ CUSTOM-WARNING-HANDLER
 (defmacro with-quiet-warnings (&body forms)
   `(let ((*all-quiet* t)
          (*saved-warnings* '()))
      (handler-bind ((warning #'quiet-warning-handler))
        ,@forms
        *saved-warnings*)))
→ WITH-QUIET-WARNINGS
 (setq saved
   (with-quiet-warnings
     (warn "Situation #1.")
     (let ((*all-quiet* nil))
       (warn "Situation #2."))
     (warn "Situation #3.")))
▷ Warning: Situation #2.
→ (#<SIMPLE-WARNING 42744421> #<SIMPLE-WARNING 42744365>)
 (dolist (s saved) (format t "~&~A~%" s))
▷ Situation #3.
▷ Situation #1.
→ NIL
```

See Also:

Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), **invoke-restart**, **muffle-warning** (*function*), **warn**

store-value *Restart*

Data Arguments Required:

a value to use instead (on an ongoing basis).

Description:

The **store-value** *restart* is generally used by *handlers* trying to recover from errors of *types* such as **cell-error** or **type-error**, which may wish to supply a replacement datum to be stored permanently.

Examples:

```
 (defun type-error-auto-coerce (c)
   (when (typep c 'type-error)
     (let ((r (find-restart 'store-value c)))
```


Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

```
       (handler-case (let ((v (coerce (type-error-datum c)
                                      (type-error-expected-type c))))
                       (invoke-restart r v))
         (error ()))))) → TYPE-ERROR-AUTO-COERCE
 (let ((x 3))
   (handler-bind ((type-error #'type-error-auto-coerce))
     (check-type x float)
     x)) → 3.0
```

See Also:

Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), **invoke-restart**, **store-value** (*function*), **ccase**, **check-type**, **ctypecase**, **use-value** (*function* and *restart*)

use-value *Restart*

Data Arguments Required:

a value to use instead (once).

Description:

The **use-value** *restart* is generally used by *handlers* trying to recover from errors of *types* such as **cell-error**, where the handler may wish to supply a replacement datum for one-time use.

See Also:

Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), **invoke-restart**, **use-value** (*function*), **store-value** (*function* and *restart*)

abort, continue, muffle-warning, store-value, use-value *Function*

Syntax:

abort &optional *condition* →|

continue &optional *condition* → **nil**

muffle-warning &optional *condition* →|

store-value *value* &optional *condition* → **nil**

use-value *value* &optional *condition* → **nil**

abort, continue, muffle-warning, store-value, use-value

Arguments and Values:

value—an *object*.

condition—a *condition object*, or **nil**.

Description:

Transfers control to the most recently established *applicable restart* having the same name as the function. That is, the *function* **abort** searches for an *applicable* **abort** *restart*, the *function* **continue** searches for an *applicable* **continue** *restart*, and so on.

If no such *restart* exists, the functions **continue**, **store-value**, and **use-value** return **nil**, and the functions **abort** and **muffle-warning** signal an error of *type* **control-error**.

When *condition* is *non-nil*, only those *restarts* are considered that are either explicitly associated with that *condition*, or not associated with any *condition*; that is, the excluded *restarts* are those that are associated with a non-empty set of *conditions* of which the given *condition* is not an *element*. If *condition* is **nil**, all *restarts* are considered.

Examples:

```
;;; Example of the ABORT retart

 (defmacro abort-on-error (&body forms)
   `(handler-bind ((error #'abort))
      ,@forms)) → ABORT-ON-ERROR
 (abort-on-error (+ 3 5)) → 8
 (abort-on-error (error "You lose."))
▷ Returned to Lisp Top Level.

;;; Example of the CONTINUE restart

 (defun real-sqrt (n)
   (when (minusp n)
     (setq n (- n))
     (cerror "Return sqrt(~D) instead." "Tried to take sqrt(-~D)." n))
   (sqrt n))

 (real-sqrt 4) → 2
 (real-sqrt -9)
▷ Error: Tried to take sqrt(-9).
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Return sqrt(9) instead.
▷  2: Return to Lisp Toplevel.
▷ Debug> (continue)
▷ Return sqrt(9) instead.
→ 3
```



```
 (handler-bind ((error #'(lambda (c) (continue))))
   (real-sqrt -9)) → 3

;;; Example of the MUFFLE-WARNING restart

 (defun count-down (x)
   (do ((counter x (1- counter)))
       ((= counter 0) 'done)
     (when (= counter 1)
       (warn "Almost done"))
     (format t "~&~D~%" counter)))
→ COUNT-DOWN
 (count-down 3)
▷ 3
▷ 2
▷ Warning: Almost done
▷ 1
→ DONE
 (defun ignore-warnings-while-counting (x)
   (handler-bind ((warning #'ignore-warning))
     (count-down x)))
→ IGNORE-WARNINGS-WHILE-COUNTING
 (defun ignore-warning (condition)
   (declare (ignore condition))
   (muffle-warning))
→ IGNORE-WARNING
 (ignore-warnings-while-counting 3)
▷ 3
▷ 2
▷ 1
→ DONE

;;; Example of the STORE-VALUE and USE-VALUE restarts

 (defun careful-symbol-value (symbol)
   (check-type symbol symbol)
   (restart-case (if (boundp symbol)
                     (return-from careful-symbol-value
                                  (symbol-value symbol))
                     (error 'unbound-variable
                            :name symbol))
     (use-value (value)
       :report "Specify a value to use this time."
       value)
```

abort, continue, muffle-warning, store-value, use-value

```
     (store-value (value)
       :report "Specify a value to store and use in the future."
       (setf (symbol-value symbol) value))))
 (setq a 1234) → 1234
 (careful-symbol-value 'a) → 1234
 (makunbound 'a) → A
 (careful-symbol-value 'a)
▷ Error: A is not bound.
▷ To continue, type :CONTINUE followed by an option number.
▷  1: Specify a value to use this time.
▷  2: Specify a value to store and use in the future.
▷  3: Return to Lisp Toplevel.
▷ Debug> (use-value 12)
→ 12
 (careful-symbol-value 'a)
▷ Error: A is not bound.
▷ To continue, type :CONTINUE followed by an option number.
▷   1: Specify a value to use this time.
▷   2: Specify a value to store and use in the future.
▷   3: Return to Lisp Toplevel.
▷ Debug> (store-value 24)
→ 24
 (careful-symbol-value 'a)
→ 24

;;; Example of the USE-VALUE restart

 (defun add-symbols-with-default (default &rest symbols)
   (handler-bind ((sys:unbound-symbol
                    #'(lambda (c)
                        (declare (ignore c))
                        (use-value default))))
     (apply #'+ (mapcar #'careful-symbol-value symbols))))
→ ADD-SYMBOLS-WITH-DEFAULT
 (setq x 1 y 2) → 2
 (add-symbols-with-default 3 'x 'y 'z) → 6
```

Side Effects:

A transfer of control may occur if an appropriate *restart* is available, or (in the case of the *function* **abort** or the *function* **muffle-warning**) execution may be stopped.

Affected By:

Each of these functions can be affected by the presence of a *restart* having the same name.

abort, continue, muffle-warning, store-value, use-value

Exceptional Situations:

If an appropriate **abort** *restart* is not available for the *function* **abort**, or an appropriate **muffle-warning** *restart* is not available for the *function* **muffle-warning**, an error of *type* **control-error** is signaled.

See Also:

invoke-restart, Section 9.1.4.2 (Restarts), Section 9.1.4.2.2 (Interfaces to Restarts), **assert**, **ccase**, **cerror**, **check-type**, **ctypecase**, **use-value**, **warn**

Notes:

```
(abort condition) ≡ (invoke-restart 'abort)
(muffle-warning)  ≡ (invoke-restart 'muffle-warning)
(continue)        ≡ (let ((r (find-restart 'continue))) (if r (invoke-restart r)))
(use-value x) ≡ (let ((r (find-restart 'use-value))) (if r (invoke-restart r x)))
(store-value x) ≡ (let ((r (find-restart 'store-value))) (if r (invoke-restart r x)))
```

No functions defined in this specification are required to provide a **use-value** *restart*.